UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Generación procedural de terreno

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Montell Serrano, Joaquim

Tutor: Vivó Hernando, Roberto Agustín

2014-2015

Resumen

Este proyecto consiste en la creación de un programa que permita tanto la edición como la visualización de un terreno, utilizando mapas de altura. Este programa ha sido creado utilizando las librerías Qt y OpenGL y se han incorporado técnicas de generación procedural para asistir a la creación de este terreno

Palabras clave: generación procedural, generación procedimental, mapa de alturas, OpenGL, Qt, diamond-square

Abstract

This project consists in the creation of a program that allows both editing and viewing a terrain, using heightmaps. This program was made using the libraries Qt and OpenGL and has been used techniques of procedural generation to assist in the creation of this terrain

Keywords : procedural generation, heightmap, OpenGL, Qt, diamond-square

Tabla de contenidos

1. Introducción

Tanto en la industria cinematográfica como en la de los videojuegos se pone mucho énfasis en crear unos escenarios realistas y creíbles. Aunque esto es fácil de conseguir en las películas con actores de carne y hueso, con una localización y decoración adecuados, para un escenario hecho por ordenador es bastante más complicado puesto que la naturaleza presenta una diversidad que es difícil de reproducir. Además, al colocar cada elemento en la escena, pueden quedar vacíos entre el suelo y el objeto dando la impresión en algunos casos de que el objeto quede flotando.

Para facilitar esta tarea, podemos utilizar la generación procedural. Es un término que hace referencia al contenido generado utilizando un algoritmo en vez de manualmente. Muchos ejemplos se encuentran en la industria de los videojuegos como son los casos de los juegos Rogue (1980), donde se construye una mazmorra cada vez que el jugador empieza la partida con el fin de que el jugador tenga una experiencia nueva con cada partida[1]; .kkrieger (2004) donde se generan las texturas y sonidos repitiendo los pasos efectuados por los artistas para reducir su tamaño[2]; Spore (2008) en el que no se podían generar las animaciones de antemano a causa de que todas las criaturas se hacen mientras se juega[3]; Left 4 Dead (2008) donde se utiliza una inteligencia artificial para adaptar la dificultad a la situación del jugador cambiando la cantidad y posición de enemigos y objetos[4]; o Borderlands (2009) que utiliza un sistema para generar los objetos consiguiendo posibilidades prácticamente ilimitada[5].

La generación procedural también ha sido aplicada a la industria cinematográfica, aunque no de forma tan extensa. Por ejemplo, en la trilogía El señor de los anillos en la que se utilizó el programa MASSIVE[1] para generar todo un ejército y el comportamiento de cada soldado[6]

1.1. Motivación

La principal motivación que ha propiciado la creación de este proyecto ha sido el interés en trabajar con la generación procedural y en aprender a mostrar un terreno realista. Asimismo, prácticamente no existen programas libres como el que se ha desarrollado que permitan un uso comercial mientras que otros tienen precios que un programador de juegos independientes[2] no se puede permitir

1.2. Objetivos

La finalidad de este proyecto ha sido crear una aplicación que permita la generación procedural de un terreno en tres dimensiones (3D) y su posterior visualización. Asimismo, se permitirá al usuario su modificación y se le dará la posibilidad de guardar el mapa de alturas[3], cargar uno ya existente o empezar desde cero

1 Multiple Agent Simulation System in Virtual Environment

2 Juegos que no cuentan con el apoyo financiero de una gran empresa

Todo esto se realizará desde una interfaz simple al estilo de un programa de dibujo donde se podrá cambiar entre el modo de visualización y el de edición mediante una tecla. Los controles definidos son los que se utilizan en los juegos en primera persona. Se han seleccionado estos por su comodidad a la hora de utilizar el teclado y el ratón a la vez.

3 Imagen en la que se almacenan datos como la elevación del terreno o, en algunos casos, el vector normal a la superficie en ese píxel.

2. Tecnologías usadas

El lenguaje elegido para el desarrollo del proyecto ha sido C++. También se barajó la utilización de Java, pero finalmente se descartó, principalmente por gustos personales y, aunque este ha sido con el que más se ha trabajado dentro del ámbito académico, C++ es el que más se ha utilizado en proyectos personales.

Las librerías utilizadas son Qt para la interfaz y openGL para mostrar la escena. El motivo de esta selección es porque, durante los estudios cursados, se han visto las bases de su funcionamiento. No obstante, en el caso de Qt, se ha trabajado con la versión para Java llamada QtJambi durante los estudios y, en el caso de OpenGL, la versión 1.0 para la cual no se aconseja el uso de la gran mayoría de sus funciones en las versiones actuales.

Por otro lado, las dos librerías son multiplataforma y son muy versátiles, lo que las hace preferible ante otras alternativas como DirectX, solo se pueden utilizar en Windows; o SDL, GLFW o GLUT que solo servirían, en el contexto de la aplicación desarrollada, para crear la ventana.

2.1. Qt

Como ya se ha dicho con anterioridad, Qt es una biblioteca multiplataforma utilizada, generalmente, para el desarrollo de interfaces gráficas de usuario. Esta mantenida tanto por Qt Company, que es la empresa creadora, y Qt Project que se encarga de coordinar a todos los individuos que están colaborando con su desarrollo.

Una gran parte de su fuerza viene dada por su sistema de señales (*signals*) y eventos (*slots*) que permite una comunicación fácil y sencilla entre los componentes de la aplicación. Esto se hace posible con un preprocesado mediante el programa qmake que completa nuestro código introduciendo todo lo necesario para el correcto funcionamiento. Unos ejemplos de esto, son el cuerpo de las señales o un sistema de introspección de tipos centrado en las señales y los eventos.

2.2. OpenGL

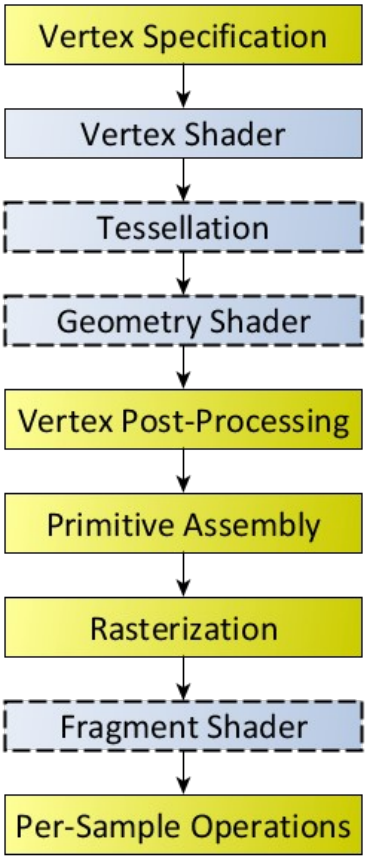
OpenGL es una especificación estándar para gráficos 2D y 3D. Esto quiere decir que, en vez de presentar una serie de funciones implementadas, tan solo describe un conjunto de funciones y la forma en la que se han de comportar dejando la parte de la implementación a los fabricantes de las tarjetas gráficas y, de esta manera, se consigue presentar una sola API[4] para todas ellas independientemente del fabricante y de las funciones que tengan aunque haya que terminar emulando alguna de ellas por *software*.

Por otro lado, OpenGL no tiene ningún mecanismo para crear ventanas. La razón de esto se da a causa de que suelen ser instrucciones dependientes del sistema. Debido a esta razón, necesitamos de una librería que cumpla esta función. Se ha elegido Qt para llenar este hueco puesto que también nos ofrece mucha facilidad para trabajar

4 Interfaz de programación de aplicaciones(del ingles: *Application Programming Interface*)

con interfaces pero, en el caso de que queramos hacer algo más sencillo, una mejor opción habría sido, por ejemplo, SDL, GLFW o GLUT.

Por último, OpenGL solo soportaba la llamada tubería fija durante la versión 1.0 lo que significa que, recibiendo unos vértices de entrada, se trataban de la misma manera dando muy poca flexibilidad. Para solucionar esto, en la versión 2.0 se introdujeron los llamados *shaders* que son unos pequeños programas que se ejecutan directamente sobre la tarjeta gráfica, modificando el tratamiento que se da a cada vértice. Durante esta versión solo estaban disponibles los *vertex shader* y *fragment shader. Estos se* encargaban de donde se dibujan los vértices y como se pintan los llamados "*fragments*"[5] respectivamente. Actualmente, existen también los *geometry shaders*, *tessellation shaders* y *compute shaders* aunque este último no se utilizará en este proyecto.

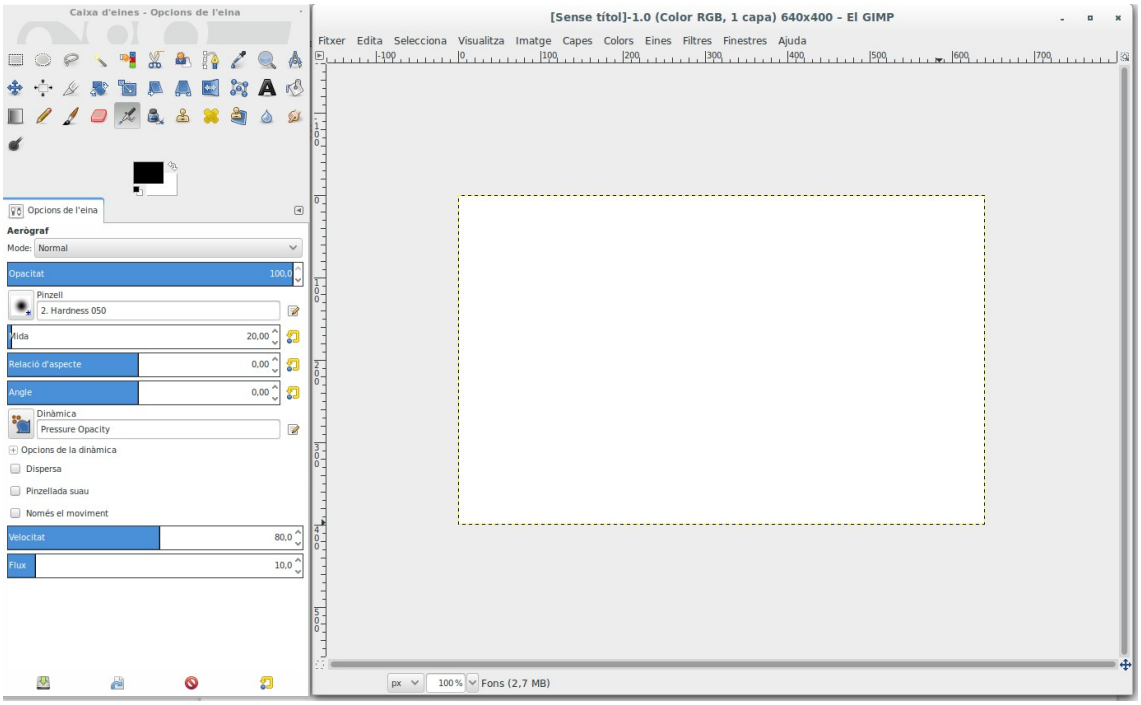


En la imagen de la izquierda se puede observar la tubería de OpenGL. Las cajas azules son elementos programables mientras que las de borde discontinuo son opcionales.[7]

5 Son elementos que contienen toda la información necesaria para dibujar un punto. La diferencia entre *fragment* y píxel reside en la posibilidad de haber varios *fragments* en un mismo píxel

3. Diseño

A causa del corto periodo de tiempo disponible y de la incertidumbre de que ampliaciones se podrían incorporar, se ha decidido hacer un diseño simple y flexible que admita la incorporación de elementos nuevos. A causa de esto, se ha decidido diseñar una interfaz al estilo del programa de dibujo GIMP donde se dispone de una caja de herramientas y el área de dibujo en dos ventanas separadas.



captura de la interfaz del GIM. Se puede observar a la izquierda la caja de herramientas con las propiedades de la herramienta seleccionada justo debajo

Teniendo esta idea en mente, se ha diseñado una interfaz donde hemos situado las distintas herramientas en la caja de herramientas separadas por secciones y, en la parte inferior de esta caja, hemos situado las propiedades de las herramientas mostrando solo las de la herramienta seleccionada. Asimismo, se ha decidido no poner una barra de menús puesto que solo se situarían las opciones de nuevo, cargar y guardar. En vez de eso, estas se han colocado junto con las herramientas.

Por otro lado, se ha decidido poner un modo de edición y otro de visualización. En el de edición se presenta el mapa de alturas y es donde está pensado que las herramientas actúen. Por otro lado, en el de visualización se muestra el terreno y se le permite al usuario desplazarse libremente por él. Como nota, habría sido interesante tener los dos modos juntos, y es lo que se pretendía en un principio, pero añadía un grado de complejidad bastante elevado con lo que se ha dejado para una futura mejora.

4. Implementación

4.1. Aprendizaje

Al principio, se intentó hacer un *framework* partiendo de un ejemplo[8] con la intención de abstraer al máximo posible la parte de dibujo. Esto fue un fracaso por intentar saltarse la etapa de aprendizaje. Debido al tiempo perdido, se decidió utilizar el que tenía el propio ejemplo aunque presentaba dos inconvenientes, en algunos casos era demasiado completo mientras que en otros era insuficiente.

De esta manera, se procedió a empezar una etapa de aprendizaje que tenía como objetivo dibujar un cubo con texturas y teselación. Esto presentó una serie de problemas puesto que la única documentación encontrada que detallaba el funcionamiento de Qt con OpenGL fue la oficial donde, generalmente, solo se explica que hace cada función pero no como se utiliza.

Después de esta introducción, procederé a explicar las etapas por las que ha pasado el proyecto durante la fase de aprendizaje. Aunque el código ya no forme parte del resultado, es una información interesante para todo aquel que esté empezando con OpenGL. No se está intentando hacer un tutorial, simplemente se intenta explicar como trabajar con estas dos librerías al mismo tiempo.

- Fase de dibujo

Las herramientas que nos presenta Qt son las clases QOpenGLBuffer, QOpenGLShaderProgram, QOpenGLVertexArrayOBject, QOpenGLContext y una que maneja algunas funciones de OpenGL que depende de la versión de OpenGL que queremos utilizar.

Lo primero que necesitamos especificar, es el contexto que vamos a utilizar. Esto lo podemos hacer mediante la clase *QsurfaceFormat* asignándole las propiedades que va a tener y luego llamar al método *setFormat* de nuestro objeto de *QOpenGLContext*.

A continuación, debemos inicializar el objeto de funciones. Para esta finalidad, tenemos el método *versionFunctions* de la clase de contexto. Esto puede fallar, por ejemplo, en el caso de que la versión que se está pidiendo no esté soportada por nuestra tarjeta gráfica, con lo que es aconsejable comprobar que no haya habido ningún error.

El siguiente paso que necesitamos hacer trata en cargar los *shaders* y compilarlos. Para ello, tenemos los métodos *addShaderFromSourceFile* si lo tenemos en un archivo o *addShaderFromSourceCode* si lo tenemos en memoria. Una vez hecho esto, tenemos que llamar al método *link para compilarlos y poder utilizarlos*.

Para terminar, nos queda enviar nuestra geometría a la tarjeta gráfica. Para ello, vamos a utilizar los llamados *vertex buffer objects*(VBO). Estos presentan la ventaja de poder enviar una gran cantidad de datos a la vez reduciendo así el número de envíos. También utilizaremos los *vertex array objects*(VAO) que nos evitan hacer una gran cantidad de llamadas cada vez que queramos pintar un VBO. Para preparar el primero, tenemos que decirle en su constructor el tipo de *buffer* que va a ser, asignarle

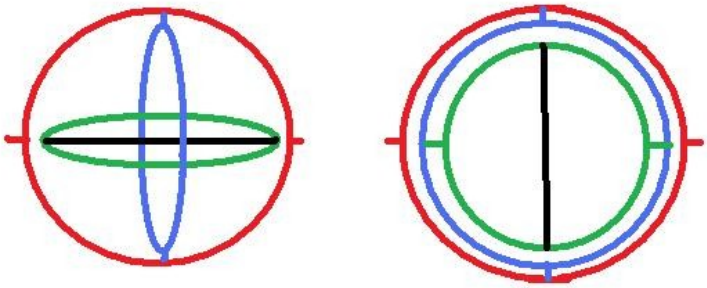
si se va a modificar o no con *setUsagePattern* y añadirle los elementos, primero, llamando a *bind* y, luego, a *allocate*. Para el segundo, tenemos que crearlo con *create*, luego, necesitamos crear un objeto de la clase *Binder con la instancia del VAO como parámetro del constructor.* Una vez hecho esto solo tenemos que activar el shader, y todos los *buffers* que queramos utilizar y podemos asignar los atributos como se haría normalmente aunque a través de los métodos *enableAttributeArray* y *setAttributeBuffer* de nuestro *shader*.

Después de haber hecho todo esto, podemos dibujar nuestro objeto simplemente volviendo a activar nuestro *shader* en caso de no estar activo, nuestro VAO y llamar a *glDrawArrays* o *glDrawElements*. Para el caso de querer utilizar texturas, Qt no nos proporciona nada por lo que se trabaja con ellas al estilo de OpenGL.

- Fase de transformaciones

Qt nos proporciona con una serie de clases para representar matrices donde ya tenemos programadas todas las operaciones con matrices. Además, tenemos métodos que multiplican la matriz actual por una de traslación, rotación o escalado como otros para construir la matriz de proyección o la de vista. Otro punto digno de mención consiste en que las matrices de rotación utilizan cuaternios con lo que nos evitamos el problema llamado G*imbal lock*.

Este problema consiste en que, al girar un eje, este puede terminar alineado con otro. Esto causa que siempre que giremos en uno de los dos ejes alineados, tendremos el mismo resultado.



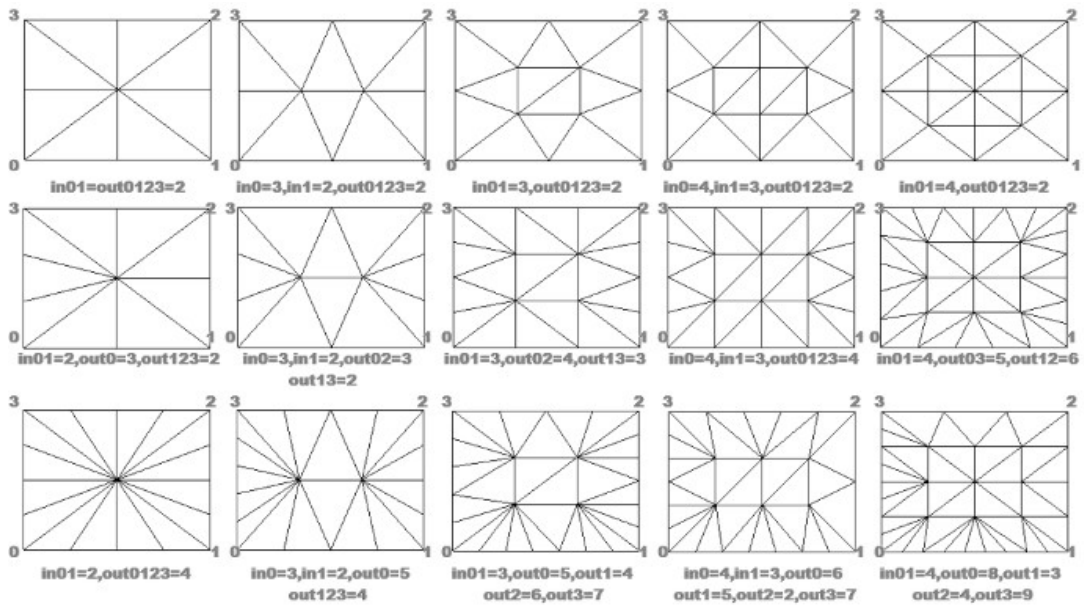
Como se puede observar en la imagen, al girar 90 º los dos ejes interiores estos quedan paralelos al exterior. A partir de este estado, girando el interior o girando el exterior el mismo número de grados, nos daría el mismo resultado provocando lo que se llama la perdida de un grado de libertad.

- Fase de *shaders*

Hasta este momento, hemos estado utilizando unos *shaders* sencillos que nos permitían dibujar los vértices con una textura pero, como queremos añadir teselación, utilizaremos tanto los *shaders* de teselación como los de geometría. Estos últimos los aprovecharemos para detectar errores pintando un borde a los triángulos.

Empezando por el *shader* de geometría, la idea que queremos seguir es calcular la distancia que hay desde un punto cualquiera hasta el borde del triángulo. Esto lo podemos conseguir fácilmente partiendo de la distancia que hay de vértice y dejar que sea interpolada por el ráster. Esta técnica fue presentada en la edición del 2006 del SIGGRAPH[9]

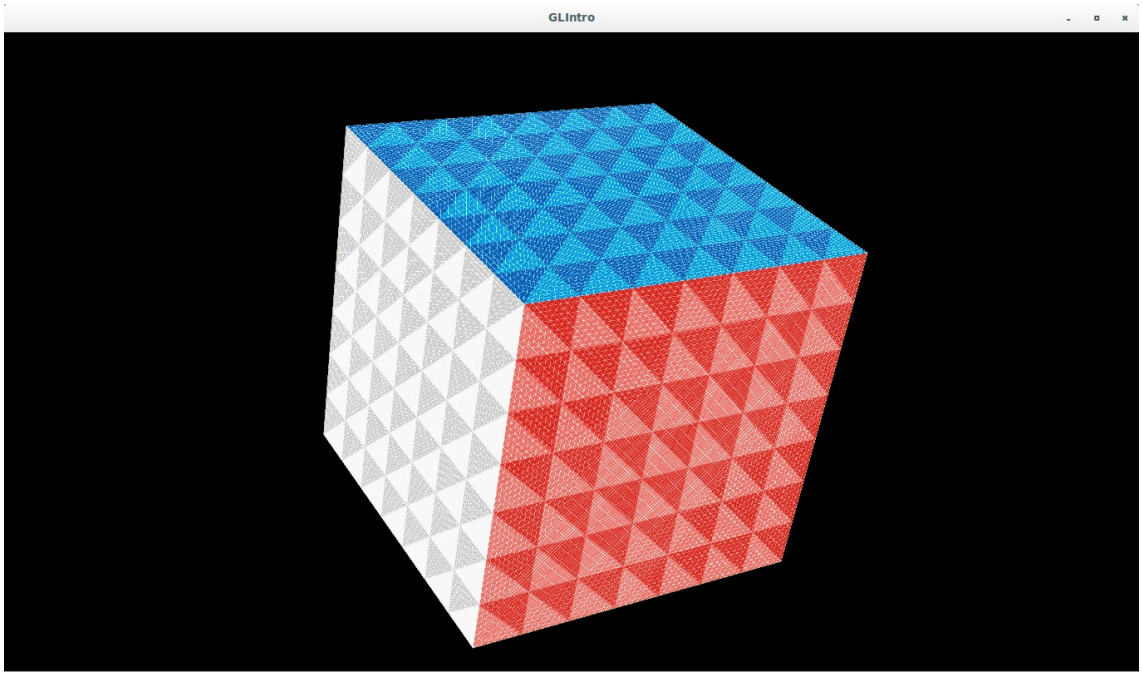
Por otro lado, la teselación nos puede servir para añadir nivel de detalle a un objeto cercano dividiendo la superficie en un número mayor de polígonos mientras que, si está lejano, esta cantidad sea menor. Para ello tenemos dos *shaders*, el de control y el de evaluación. El primero nos sirve para determinar cuanto detalle le queremos dar mientras que, el segundo, nos sirve para posicionar los polígonos generados que pueden ser triángulos, cuadrados o isolíneas. Para determinar su cantidad utilizaremos dos vectores de reales en el *shader* de control que son *gl_TessLevelOuter (*out*)* y *gl_TessLevelInner* (*in*). El primero tiene 4 elementos mientras que el segundo solo 2, sin embargo, no todos son utilizados. Esto depende del tipo de primitiva, utilizándose todos para el rectángulo, solo uno de *in* y tres de *out* para el triángulo y solo dos de *out* para las isolíneas.



En la imagen podemos ver una serie de rectángulos a los que se les han aplicado distintos niveles de teselación. También se puede observar que cada valor de *out* corresponde al número de triángulos de cada lado aunque, estos lados, están nombrados en orden inverso al esperado, es decir, el lado 0 es el 0-3, el 1 es el 3-2 y sucesivamente.

En cuanto a *in*, simplemente se trata del nivel horizontalmente para 0 y verticalmente para 1. Esto es más fácil de ver si intentamos partir el rectángulo en filas y columnas consiguiendo, para el primero, dos filas y dos columnas. En el segundo, dos filas y tres columnas…

- Resultado



En la imagen se puede encontrar el resultado del aprendizaje. Este programa permite la visualización de un cubo animado que gira sobre los ejes x e y. También tiene aplicada la textura que se puede ver, donde todas las caras tienen el mismo patrón pero de un color distinto. Además, se le ha aplicado un nivel de teselación fija a todas las caras que se puede observar por las líneas blancas que hay sobre él, aunque no se puede terminar de apreciar los triángulos a causa de la gran cantidad de ellos.

En este punto, se ha decidido crear una clase para representar de forma abstracta un objeto. La idea principal de esto ha sido, como se pretende tener varios modelos en un futuro donde cada uno puede tener su propio *shader*, la de tener un sitio donde se pueda guardar cual esta activo y, de esta manera, saber cual está activo para solo cambiarlo si no es el que necesitamos.

4.2. Renderización del terreno

Para este primer paso hacia el programa final, hemos reutilizado gran parte del código que teníamos hecho para el cubo. Los cambios principales han sido respecto a la definición de la geometría aunque, la gran modificación, ha sido respecto a los *shaders*.

Para empezar, hemos definido nuestro VBO formando una malla donde hemos prescindiendo del valor de la altura. Esto se ha hecho así porque queremos asignárselo durante la teselación. Asimismo, le hemos asignado tres texturas, sin contar el mapa de alturas, para darle diversidad aplicando, en la mayoría del terreno, la textura de hierba mientras se utiliza una textura de piedras en las pendientes con una inclinación pronunciada. Por último, se ha utilizado una textura de nieve para os puntos que tiene una altura mayor que un valor fijo.

En la parte de *shaders*, se decidió tomar la longitud final de los lados cuando estén pintados en la pantalla como un primer intento de determinar el nivel de teselación . Para conseguir esto, tenemos que convertir cada vértice al espacio de proyección y dividir cada componente del vértice por el cuarto y calcular la distancia entre ellos. Para esto necesitamos la matriz del modelo, la de vista y la de proyección.

La primera consiste en las transformaciones acumuladas que se han efectuado sobre el objeto con lo que, al multiplicarla por cada vértice, conseguimos transformarlos del espacio local o del modelo al espacio del mundo. Esto nos es realmente útil puesto que nos permite modelar un objeto en el origen y duplicarlo cuantas veces queramos situándolo en distintos lugares.

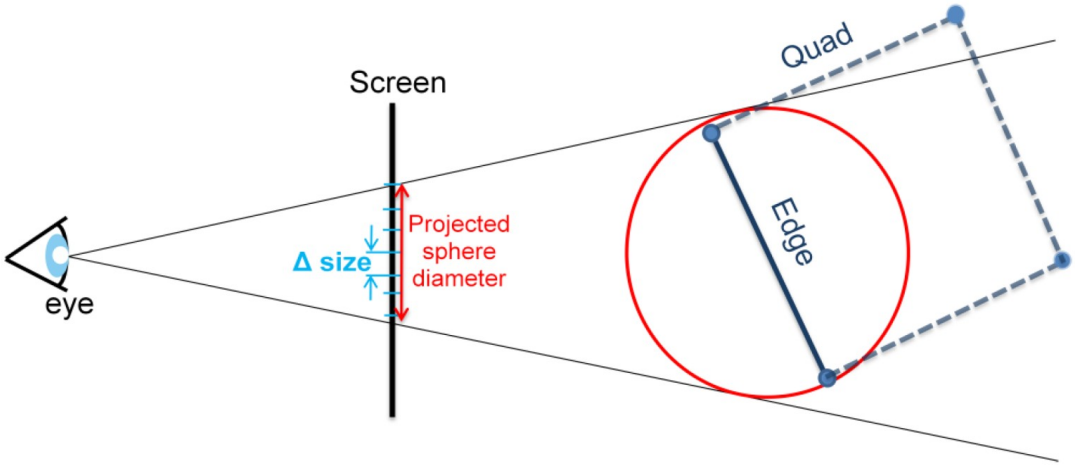
Por otro lado, la matriz de vista es similar a la del modelo aunque, si la primera es para mover un objeto por el mundo, esta sirve para mover el mundo. Esto se basa en la idea de que, si nos movemos en un vehículo, lo que nosotros observamos que es el mundo es el que se mueve mientras nosotros nos quedamos quietos

Por último, la matriz de proyección es la que se encarga, puesto que estamos utilizando perspectiva, de que los objetos que estén más lejanos se vean más pequeños. Esto se consigue modificando el cuarto componente de los vértices, también llamado w, dándole un valor mas alto cuanto más lejos.

Regresando al tema de la teselación, esta aproximación presentó dos inconvenientes. El primero se da por la propia naturaleza de la teselación. Puesto que estamos dividendo polígono, la posición de los vértices no se mantiene siempre igual con lo que, cuando accedemos al mapa de alturas, la altura resultante no siempre es la misma en el mismo punto de la escena. Estas dos cosas provocan que, en algunos puntos, de la impresión de un temblor al acercarse o alejarse de estos puntos.

El segundo problema se da por algo bastante más sencillo y es más fácil de ver. Este ocurre cuando uno de los polígonos está situado en uno de los planos que sean paralelos, o se acerquen a serlo, al vector que representa hacia donde se está mirando, lo que provoca que, al lado, se le asigne un nivel de detalle bastante menor que el que debería tener.

A causa de esto se decidió buscar un algoritmo más desarrollado que evitase el segundo problema y, de ser posible, que mitigue o elimine el primero. Con esto en mente, se encontró[10] un método parecido al primero, pero que determinaba el nivel de detalle a partir la proyección de una esfera en vez de la del lado directamente. Para este método, calculamos el punto medio del lado obteniendo el primer punto. A continuación, desplazamos este punto verticalmente en una cantidad igual a la longitud del lado para obtener el segundo punto y utilizamos estos dos puntos para calcular el nivel de detalle. Este sistema se puede observar en la siguiente imagen.



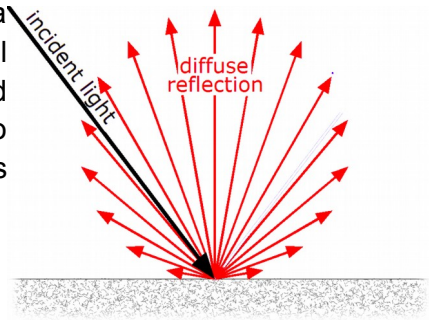
Este cambio consigue eliminar totalmente el segundo problema y la parte que causaba del primero con lo que solo nos queda del primer problema la parte causada por los cambios de altura bruscos. Una opción para mitigar esto podría ser cambiar la función para que alcance el máximo antes, de forma que este temblor se dé a una distancia donde no se perciba.

Cambiando de tema, para el renderizado del terreno hemos utilizado tres texturas como ya hemos dicho con anterioridad. Como no disponemos de un artista para hacerlas, hemos utilizado las del ejemplo.

Por otro lado, hemos utilizado el modelo de Phong para el sombreado. En este modelo, el color final viene dado por la suma de tres componentes: la ambiente, la difusa y la especular.

La luz ambiente intenta imitar la luz que se ha reflejado de las distintas superficies del entorno. Esta luz incide igualmente en todas las superficies y viene dada por la ecuación $I = I_a \cdot k_a$ donde I_a es la intensidad de la luz ambiente mientras que K_a es el coeficiente de reflexión del objeto que depende de las propiedades de su material.

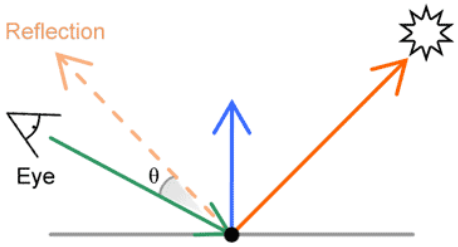
La luz difusa intenta imitar a la luz reflejada por una superficie de manera adireccional. Esta luz depende del ángulo de incidencia dando la máxima intensidad cuando los rayos de luz son perpendiculares. Con todo esto en mente, podemos deducir que su formula es $I = I_L \cdot k_d \cdot \cos(\Theta) = I_L \cdot k_d \cdot (\vec{N} \cdot \vec{L})$ donde:



- I_L es la intensidad fuente
- Θ es el ángulo incidencia
- k_d es el coeficiente reflexión difusa del objeto
- N es la normal de la superficie
- L es el vector de la dirección de la iluminación

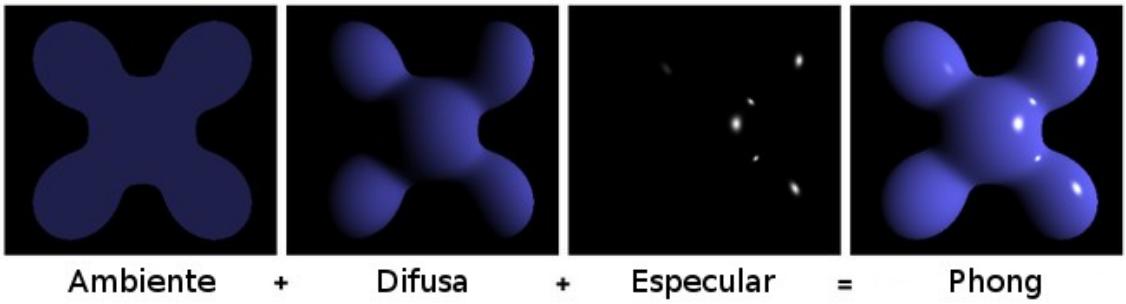
  Tanto N como L tienen que estar normalizados, es decir, tienen que tener una longitud de valor uno.

La reflexión especular se trata del reflejo que produce una luz al iluminar un objeto liso que será mayor cuanto más pulida sea su superficie. Esta contribución al color dependerá del color de la luz más que del objeto y será máxima cuando el ángulo de reflexión coincida con el angulo de visión.

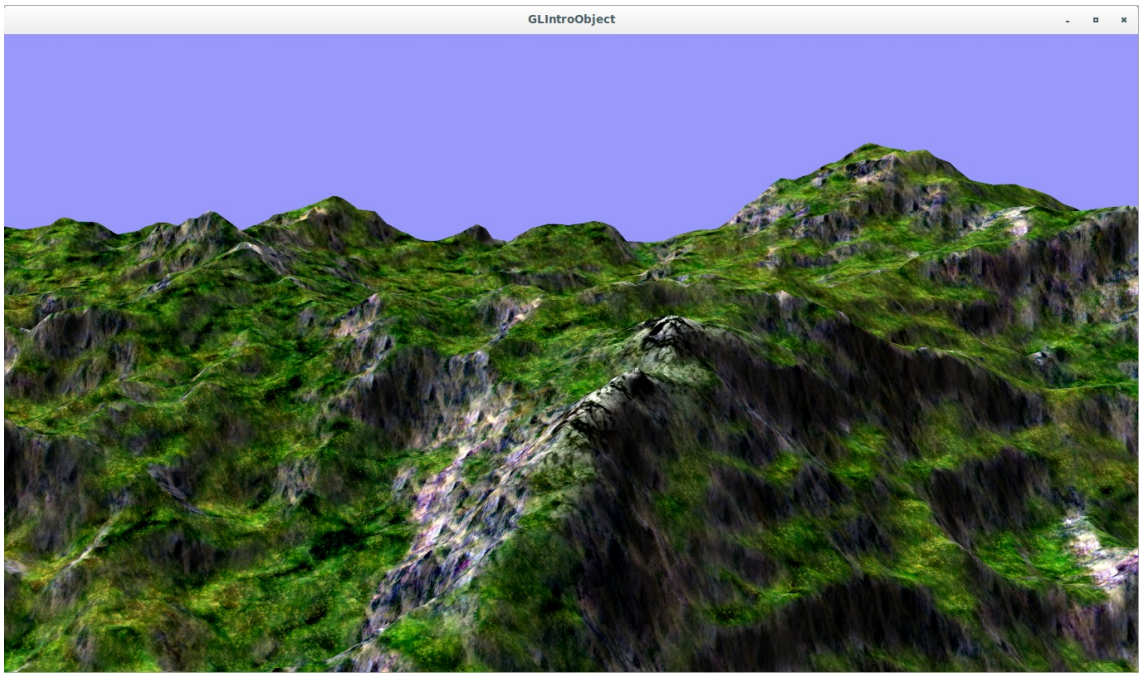


Concluyendo, la formula será $I = I_L \cdot k_s \cdot \cos^n(\Theta) = I_L \cdot k_s \cdot (\vec{R} \cdot \vec{V})^n$ donde:

- k_S es el coeficiente de reflexión especular.
- Θ es el ángulo entre R y V.
- n es el coeficiente de especularidad.
- V es el vector de dirección de la vista
- R es el vector de reflexión y se calcula de la siguiente forma: $\vec{R} = 2 \cdot \vec{N} \cdot (\vec{N} \cdot \vec{L}) - \vec{L}$



Aplicando las técnicas anteriores hemos conseguido el siguiente resultado:



4.3. Editor

Para hacer las herramientas de diseño, como se comentó con anterioridad, se ha escogido trabar en 2D por su simplicidad, mostrando el mapa de alturas y, luego, posibilitando la visualización del terreno en 3D en un modo distinto. También se han añadido un conjunto de herramientas reducido que, aunque sea menor en número que las que tendría un programa de estas características, ya posibilitan la edición del terreno en la misma medida.

Para la presentación del mapa de alturas, hemos creado una superficie en el origen para, de esta forma, controlar el *zoom* que se hace acercándonos o alejándonos en el eje z. Asimismo, se ha decidido controlar la posición con las mismas teclas que en la visualización del terreno en vez de con las convencionales barras de desplazamiento.

Para saber en que lugar del mapa de altura se ha pulsado y, de esta forma, poder realizar las distintas tareas, necesitemos utilizar técnicas típicas del trazado de rayos. Antes de nada, vamos a definir que es un rayo.

Un rayo está representado, en forma vectorial, por un punto para indicar el lugar de salida y un vector para la dirección. En esencia, un rayo consiste en la linea con origen en el punto dado que atraviesa toda la escena.

Uno de sus usos, el que nos interesa, es el de detectar colisiones entre, por ejemplo, una bala y cualquier elemento de la escena. Para ello, se traza un rayo en el origen de la bala y se busca la intersección con el volumen que delimitan los objetos de la escena, típicamente cajas o esferas, y elegimos la más cercana al punto de origen (en el caso de que solo queramos la primera colisión).

En nuestro caso, vamos a querer encontrar que punto del plano formado por los ejes x e y, puesto que es donde vamos a tener todos los objetos, es el que está debajo del ratón. Para ello, partiremos el punto donde esta la cámara situada y, para el vector de dirección, vamos a tener que convertir el punto donde está situado el ratón en la pantalla a las posiciones del mundo. Esto nos presenta un problema puesto que, mientras que lo que se muestra por la pantalla tiene dos dimensiones, el mundo está representado en tres.

Para solucionarlo, puesto que las coordenadas de la pantalla en OpenGL tienen un rango de -1 a 1, incluyendo la profundidad, podemos elegir como tercer componente el punto más lejano, es decir, para los dos primeros componentes, utilizaremos la posición del ratón que habremos convertido antes al rango de OpenGL mientras que para la profundidad escogeremos -1.

Una vez tenemos el punto en el espacio de la pantalla, tenemos que convertirlo al espacio del mundo. Para ello, tenemos que efectuar las transformaciones invertidas que realizamos con anterioridad para pasar de la posición del mundo a la de la pantalla, esto es multiplicar el punto por la inversa de la matriz de proyección y, luego, por la inversa de la matriz de vista.

En estos momentos, tenemos tanto el punto de origen como el vector de dirección aunque el vector tiene una longitud igual a la distancia que hay entre la cámara y el plano más alejado de la vista por lo que se debería normalizar.

Para terminar, solo tenemos que detectar cuando se produce la intersección con el plano deseado y sustituir en la ecuación para sacar la posición en el mundo tal que la profundidad valga cero.

4.4. Herramientas

- Nuevo, guardar y cargar

Estas tres acciones han sido las más sencillas de programar y, por ello, las primeras. Para la función de nuevo, solo ha habido que reinicializar la escena cargando sus valores de base.

Para guardar y cargar, al solo disponer de una sección de terreno y de tamaño fijo, se ha decidido utilizar archivos de imagen sueltos. De esta forma, se ha usado las funciones de Qt para mostrar diálogos para elegir los ficheros destinatarios de la acción. Estos ficheros son simplemente imágenes que contienen el mapa de alturas.

Aunque permite cargar cualquier tamaño de imagen, solo está pensado para trabaja en imágenes del tamaño base, es decir, de 1024 por 1024 píxeles, teniendo la posibilidad de dar problema con algunas funciones si se carga una imagen de tamaño distinto.

- Herramienta de selección

En este punto, se ha creado una herramienta para delimitar el área en la que las otras herramientas hacen efecto. Se ha decidido que esta área tenga forma rectangular aunque, por esta razón, su utilidad es limitada puesto que las líneas rectas no se suelen encontrar en la naturaleza.

Una mejor elección habría sido darle forma de elipse, esto podría incrementar sensiblemente su utilidad, pero al ser una forma fija, los casos en los que se ajuste a la forma que el usuario quiera también serán limitados por lo que se decidió no incluirlo en la primera versión.

Otra opción más válida, podría ser la construcción del área a partir de una serie de puntos definidos por el usuario e incluso se podrían convertir en, por ejemplo, curvas de Bézier para eliminar las líneas rectas. Esta idea se descartó dada la complejidad a la hora de dibujarla y de como de tratarla.

- Herramientas de pintura

Para desarrollar la herramienta de dibujo, se ha elegido crear algo similar a la herramienta de pincel de un programa de dibujo como el GIMP. Esta herramienta, tiene la función de asignarle una elevación determinada a los puntos que estén dentro de un radio alrededor del punto donde se encuentra el ratón. Además, dispone de un ajuste para darle un suavizado e intentar evitar, en caso de que así se desee, la creación de acantilados.

Esto no ha dado los resultados que se esperaba con lo que resulta fácil crear acantilados mientras que para crear laderas hay que ir creando una serie de capas

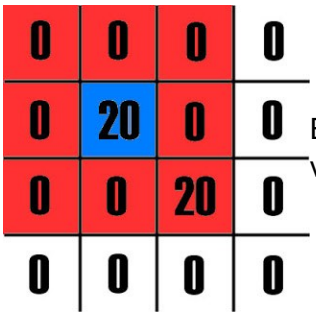
con un incremento suave en la elevación. Como esto es tedioso, habría que incluir una nueva herramienta para elevar o reducir la altura.

También sería interesante disponer de una herramienta de llenado que le asignase una altura a toda un área, aunque pierde gran parte de su utilidad sin poder seleccionar una forma más dinámica.

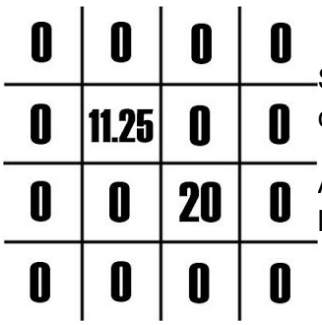
4.5. Filtro de suavizado

Se ha decidido desarrollar un filtro de suavizado para, por una parte, facilitar la creación de laderas que, como ya se ha comentado, no se ha conseguido la facilidad que se quería en un principio y, por otra parte, porque los algoritmos de generación pueden proporcionar un terreno que nos interese pero con un aspecto demasiado caótico para ser utilizado.

El algoritmo implementado trabaja intentando igualar cada píxel de nuestro mapa de alturas al de sus vecinos. Para ello, calcula la media de los vecinos de cada píxel y luego le asigna la media entre el resultado y su valor. Un ejemplo de la ejecución podría ser como sigue:

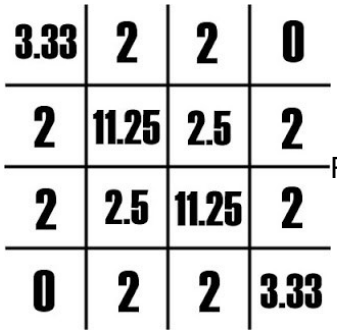


Estado inicial, queremos suavizar el valor en azul. Sus vecinos salen pintados en rojo.



Sumamos el valor de sus vecinos y lo dividimos por su cantidad: 20/8=2,5.

Ahora sacamos la media del resultado anterior con el valor de la casilla: (20+2,5)/2=11,25.



Resultado de aplicar este proceso a todas las casillas.

4.6. Generación de terreno

Una parte importante de este proyecto ha tratado de la generación procedural de terreno. Para ello se ha utilizado una técnica que se basa en la autosimilaridad de los fractales.

La autosimilaridad es una propiedad que se da en un objeto cuando todo él es igual o parecido a una parte de sí mismo. Por otro lado, un fractal es un objeto que tiene una estructura que se repite a varias escalas.

Como ejemplo de autosimilaridad, podemos poner el esqueleto de una hoja donde podemos observar un tallo grueso que recorre el centro de la hoja y del que salen tallos más finos. Si nos acercamos a uno de estos, también observamos más tallos que presentan esta misma estructura. En la naturaleza se pueden observar un sinfín de ejemplos como este.

Como contraejemplo, podemos hablar de una esfera. Si tomamos una esfera muy grande y nos vamos acercando gradualmente a ella, llegará un momento en el que vamos a dejar de ver que es una esfera

El terreno también tiene esta propiedad, por ejemplo, podríamos encontrar una piedra que tenga una silueta similar a las montañas que se ven en el horizonte independientemente de la escala en la que se encuentren.

- Algoritmo del desplazamiento del punto medio

Este es un algoritmo que genera algo similar a una silueta de unas montañas. Aunque sea un algoritmo que trabaja sobre dos dimensiones, es interesante su explicación puesto que, el utilizado para el proyecto, se podría ver como una implementación en tres dimensiones de este.

Este algoritmo consiste en, partiendo de una linea, encontrar el punto medio y desplazarlo una cantidad aleatoria. En este momento, reducimos el rango de los números aleatorios en cierta cantidad y repetimos con los dos nuevos segmentos. Esto se hace un número determinado de veces.

Poniendo un ejemplo, vamos a empezar con una linea, buscamos el punto medio, calculamos el valor medio y le sumamos un valor aleatorio:

Ahora que tenemos tres puntos, reducimos el rango que pueden tener los números aleatorios y hacemos la misma operación en cada segmento:

Cuanto más repitamos este proceso, conseguiremos un resultado menos poligonal:

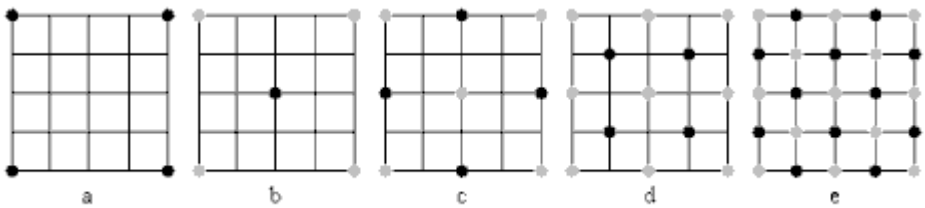
Como se puede observar la altura máxima viene dada por el rango inicial de los números aleatorios más la media de los extremos. El método se puede modificar para hacer que los dos extremos reciban también un valor aleatorio. De esta forma, evitamos tener que asignarles un valor fijo pero, a causa de esto, no podremos encadenar varias ejecuciones sin modificar la altura de todos los puntos.

En cuanto a la cantidad en la que reducimos el rango, este valor afectará a lo escarpado que será el resultado siendo más suave cuanto mayor se reduzca en cada iteración. Este valor suele recibir el nombre de dureza (*roughness* en ingles) .

- Algoritmo diamond-square

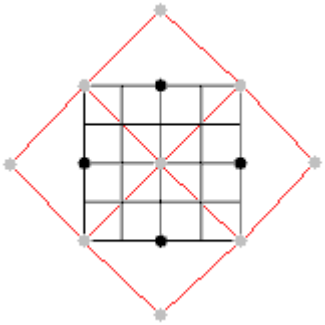
Este algoritmo, como se ha comentado en el apartado anterior, tiene un funcionamiento similar al del desplazamiento del punto medio. También hay que buscar el punto medio y desplazarlo verticalmente en una cantidad aleatoria que está dentro de un radio que se va reduciendo con cada iteración.

La diferencia entre este y el anterior, aparte del espacio en el que trabajan, se basa en el método para encontrar el punto medio. El primero simplemente utiliza la posición situada entro los dos puntos mientras que, en este, se dan dos casos. El primero de ellos, es buscar el centro, donde las dos diagonales se encuentran, de cada cuadrado mientras que, el segundo, trata de encontrar el centro de cada diamante o rombo generado en el paso anterior. Estos pasos se llaman *diamond* y *square* respectivamente.



En la imagen anterior, se puede observar la serie de pasos que se harían para una imagen de 5x5 donde se marca los puntos generados en el paso actual en negro. En el primero se da el estado inicial. Aquí tenemos que darle unos valores iniciales a las 4 esquinas para poder empezar. Luego, los pasos b y c corresponden a la primera iteración mientras que los d y e a la segunda.

Este algoritmo presenta la necesidad de utilizar una rejilla que tenga un tamaño equivalente a una potencia de dos más uno. Esto es causado por las sucesivas divisiones hechas para sacar el punto medio y, como se necesita tener un punto entre los dos originales, este número tiene que ser impar.



Asimismo, presenta un problema cuando los puntos se encuentran en los bordes puesto que, en la segunda fase, necesitamos un punto de los 4 que forman el diamante que cae fuera del área como se puede observar en la imagen. Tres opciones para evitar este problema consisten en:

- Utilizar un valor similar a los que se dan en los otros tres puntos.
- Solo utilizar tres puntos para calcular la media.
- Utilizar los valores que se dan en la parte opuesta, es decir, tratar los puntos de la misma manera que tener otra imagen como la que se está trabajando pegada al lado.

Este método es criticado por Gavin S. P. Miller[11] por generar de forma artificial una montaña en el centro. Sin embargo, no se ha podido observar este fenómeno en nuestro proyecto llegando a la conclusión de que Miller le asignaba una elevación grande al punto central llegando a tener estos resultados.

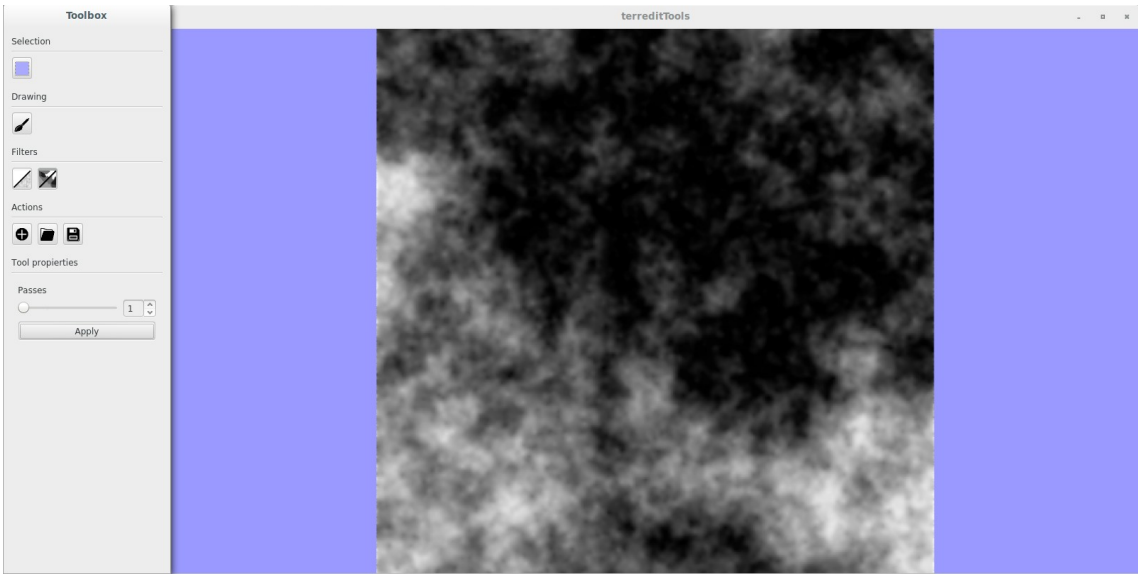
Este algoritmo ha sido el que se ha utilizado para la aplicación, principalmente, por dos razones. Una de ellas es su simplicidad, teniendo un método fácil de entender y que resulta muy visual. La segunda razón fue que puedes asignar un valor dado a determinados puntos, siempre y cuando no sean puntos generados por las últimas iteraciones, y el algoritmo seguirá generando un terreno realista mientras que tiene en cuenta estos valores dados. Sin embargo, en el programa final esta propiedad no se ha podido utilizar dada la complejidad que presentaba darle al usuario, que no tenía porque saber las limitaciones del algoritmo, la opción de elegir los puntos de control. Para suplir esta carencia, se ha decidido dar la posibilidad de dibujar un terreno y proyectar encima el resultado de la generación.

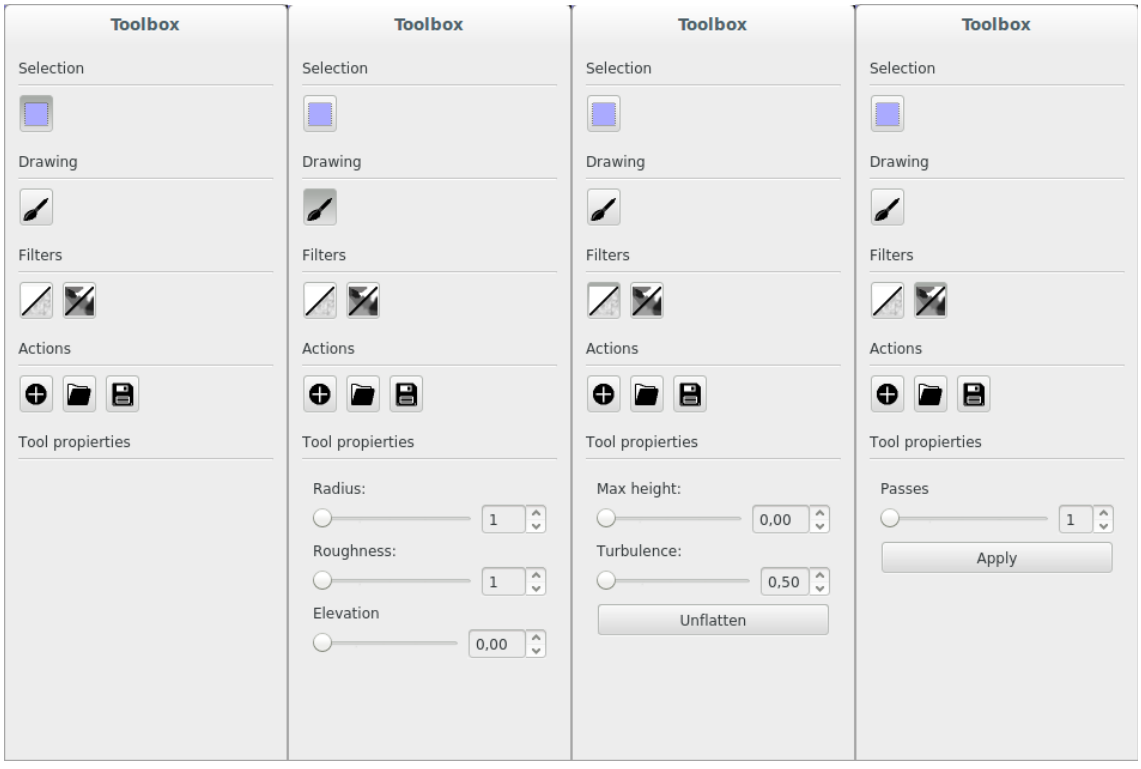
- Otro algoritmo: *simplex noise*

Otro algoritmo que se ha tenido en cuenta a la hora de desarrollar el programa final ha sido el *simplex noise*. Esta técnica es una mejora respecto a otra anterior llamada *Perlin's noise* y, aunque sus usos están centrados en dar realismo a los gráficos generados por computadora, también se puede utilizar como generador de terreno como ha sido el ejemplo del famoso juego Minecraft[12].

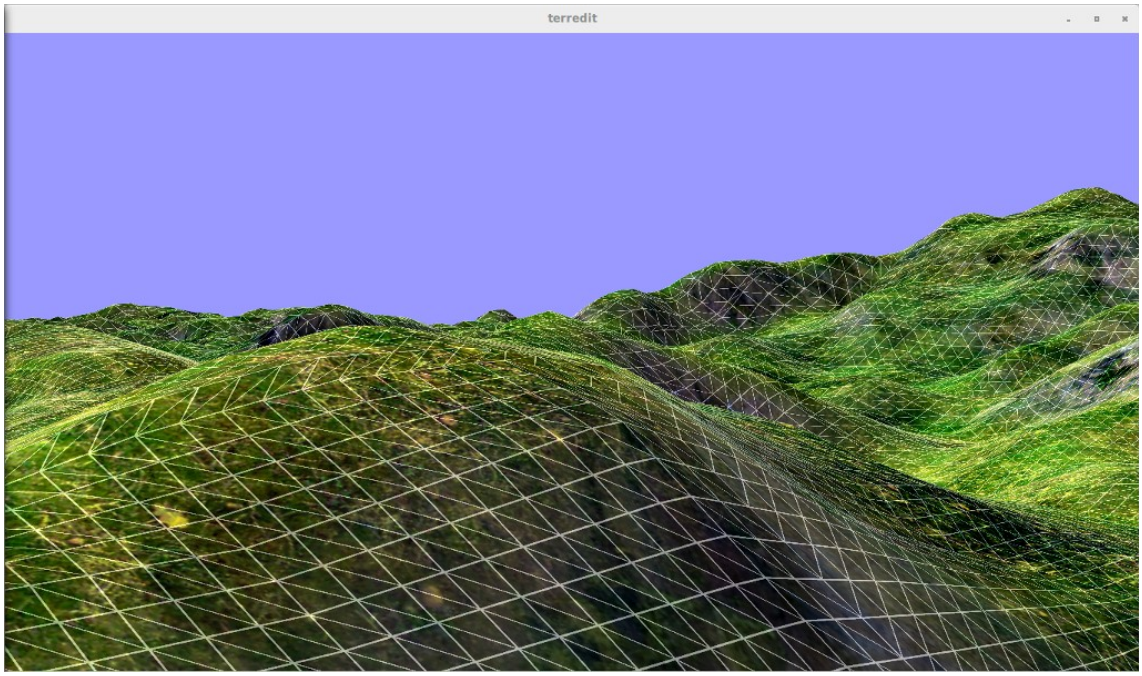
Debido a la complejidad de este algoritmo y que no se haya llegado a entender su funcionamiento, se ha decidido descartar su implementación aunque seria interesante disponer de los dos algoritmos en la aplicación para poder comparar los resultados.

4.7. Resultado final





Caja de herramientas con las propiedades de cada una



La última imagen se ha tomado para mostrar la teselación y como se va reduciendo la cantidad de triángulos a medida que se aleja del observador. Otro dato interesante es que, efectivamente, los triángulos tienen un tamaño similar independientemente de la distancia al observador.

Por último, para hacerse una idea de la potencia de la teselación, necesitamos dibujar más de 104 millones de triángulos para conseguir el mismo nivel de detalle si no disponemos de esta. Sin embargo, con ella conseguimos tener una tasa de fotogramas suave con una escena de estas dimensiones.

5. Posibles mejoras y trabajo futuro

En este apartado, vamos a hablar sobre las posibles mejoras que se han pensado que se podrían desarrollar. Asimismo, se ha decidido separarlas en dos grupos: las que se han tenido en cuenta a la hora de desarrollar el código y las que no. Esta separación no implica que unas no puedan o sean más costosas de hacer, simplemente es para diferenciar las que han surgido durante el desarrollo de las se tenía pensado en incluir si se disponía de más tiempo.

5.1. Mejoras previstas

- Capacidad de ampliar el terreno a través de parches

La idea detrás de este concepto trata en tener la escena compuesta de cuadrados más pequeños o de un tamaño configurable pero global, es decir, que todos tengan el mismo tamaño.

Actualmente el terreno está representado por un atributo, solo tenemos para almacenar uno de estos cuadrados. Para aplicar esta idea, tendríamos que modificar esto de tal forma que utilizásemos algún contenedor para almacenar todos los cuadrados, como podría ser una tabla hash indexando por la posición.

Por otro lado, tendremos que guardar todos los mapas de alturas con algún sistema para identificarlos. Mi propuesta sería darles un nombre a cada archivo que representase la posición y añadir un archivo con la configuración del mundo.

Otro aspecto que sería afectado considerablemente por este cambio, serian los filtros que se han utilizado. El menos afectado de los dos sería el de suavizado. La implementación de este algoritmo ignora los bordes para evitar roturas entre estos cuadrados, pero como contraefecto deja unos muros muy pronunciados en las juntas. Para hacer correctamente esto, habría que utilizar los valores de cada uno de los parches vecinos para el suavizado.

En cambio, la generación del terreno se ve mucho más afectada. Para el primer cuadrado, se puede disponer del algoritmo tal como está, pero en los siguientes hay que tomar en cuenta los valores de los cuadrados circundantes. Esto puede sacar a relucir todo el potencial del algoritmo implementado puesto que se puede controlar de forma local que puede contener el cuadrado a través de asignarle unos valores a las esquinas junto con la limitación de la altura. Por ejemplo, si queremos hacer unas montañas al lado de una pradera, podemos asignarle al cuadrado donde estarán las montañas una altura elevada con un valor de dureza medio-alto mientras que, el que contiene la llanura, darle una altura baja con una dureza baja. Asimismo, si quisiéramos añadir un “mar”, tan solo tendríamos que forzar las esquinas de donde va a parar el mar a una altura que esté por debajo del nivel del mar e ir añadiendo más cuadrados con las mismas propiedades que para la llanura.

- Incluir el concepto de nivel del mar así como un sistema para mostrar líquidos

El agua es un componente que le da mucho realismo a una escena, aunque sea una implementación sencilla. El concepto clave para la implementación que se tiene en mente viene dado por el hecho de que si, haces un agujero que llegue al nivel del mar, vas a encontrar agua. En la realidad, esto pasa si solo hay materiales porosos desde donde haces el agujero hasta el mar, pero para una primera aproximación da unos resultados aceptables.

Para la representación del agua, se ha pensado en utilizar una malla lisa a la que aplicarle un desplazamiento similar al que se le hace con el terreno pero en una escala menor y añadirle un desplazamiento relativo respecto el tiempo para simular el oleaje.

- Incluir objetos

Otra herramienta interesante seria la de posibilitar la inclusión de modelos al editor y colocarlos por el terreno. Pensando en esta utilidad, se desarrolló una clase para representar objetos abstractos. De esta forma, podemos tener una lista de estos objetos en nuestra escena y dibujarlos sin tener que preocuparnos de si son un árbol o un espejo.

5.2. Mejoras no previstas

- Herramientas elevar y disminuir la altura

Como ya se ha comentado, la herramienta de dibujo no ha dado el resultado que se esperaba con lo que ha surgido la necesidad de unas herramientas más específicas. Estas herramientas deberían facilitar la creación de un terreno más suavizado a partir de un incremento en la altura en vez de la altura final.

- Aprovechar la imagen

En estos momentos, estamos utilizando una imagen de en formato ARGB. Esto quiere decir que la imagen tiene 4 componentes almacenados en un byte cada uno. Estos componentes representan la cantidad de rojo, de verde, de azul y lo “transparente” que es el color final.

Como estamos representando una altura en cada píxel, solo utilizamos uno de los tres componentes del color y lo replicamos a los otros dos. Esto nos está limitando a tener 256 alturas distintas que suele ser bastante para muchos casos. Sin embargo, el algoritmo de generación está pensado para trabajar en un rango de enteros o, mejor aun, de reales teniendo un resultado más aplastado del que debería tener para los valores dados.

Puesto que solo estamos utilizando un componente de la imagen, es decir, estamos utilizando una cuarta parte de la capacidad de la imagen. Para aprovechar toda la imagen podríamos partir un entero de 32 bits para almacenarlo en cada uno de los cuatro componentes consiguiendo un rango mucho más amplio.

6. Conclusiones

El desarrollo de este proyecto ha supuesto un primer contacto con la programación de gráficos moderna que, aunque haya sido duro, ha sido muy interesante y me ha servido para suplir una carencia que creo que tiene el grado.

Durante la realización de este proyecto, se han utilizado herramientas muy potentes como son Qt y OpenGL pero que presentan sus inconvenientes, sobre todo, OpenGL donde un error en un *shader* puede provocar que no se pinte nada y sin posibilidad de ver que es el lo que falla puesto que no tiene ninguna herramienta que de información sobre las variables.

El proyecto ha consistido en la implementación de un programa que permite la creación, edición y visualización de un terreno tridimensional utilizando unas herramientas simples de edición. Además, se han utilizado unos filtros para hacer un suavizado, consiguiendo facilitar la creación de un terreno más suavizado, y para generar el mapa de alturas, utilizando el algoritmo diamond-square

El tiempo del que se ha dispuesto ha sido bastante reducido a causa de que las asignaturas han consumido más tiempo de lo previsto. Aun así, se ha conseguido alcanzar unos objetivos mínimos obteniendo una aplicación totalmente funcional y con un mínimo de errores.

Por último, conocimientos dados en las asignaturas de análisis matemáticos y álgebra han sido de gran utilidad para entender todas las bases matemáticas que tiene OpenGL y han sido de gran ayuda para los filtros implementados. Asimismo, las asignaturas de introducción a sistemas gráficos interactivos e interfaz persona-computador hacen una buena base para las librerías utilizadas.

[1] Glenn R Wichman, A brief history of “Rogue” [en línea]. http://www.wichman.org/roguehistory.html [Último acceso: 23 de agosto de 2015]

[2]proyecto .kkrieger [en línea]. http://www.pouet.net/prod_nfo.php?which=12036 [Último acceso: 23 de agosto de 2015]

[3]Spore creature editior hands-on [en línea]. http://web.archive.org/web/20060614101226/http://www.gamespot.com/pc/strategy/spore/news.html?sid=6150118 [Último acceso: 23 de agosto de 2015]

[4]Michael Booth, Valve, The Ai systems of Left 4 Dead [en línea] http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf [Último acceso: 23 de agosto de 2015]

[5]Ryan Kuo, Why Borderlands 2 Has the Most Stylish Guns in Gaming [en línea]. http://blogs.wsj.com/speakeasy/2012/04/19/why-borderlands-2-has-the-most-stylish-guns-in-gaming/ [Último acceso: 23 de agosto de 2015]

[6]Massive software [en línea] . http://www.massivesoftware.com/about.html [Último acceso: 24 de agosto de 2015]

[7] OpenGL, Rendering Pipeline Overview [en línea]. https://www.opengl.org/wiki/Rendering_Pipeline_Overview [Último acceso: 25 de agosto de 2015]

[8]Sean Harmer, OpenGL in Qt 5.1 – Part 5 [en línea] http://www.kdab.com/opengl-in-qt-5-1-part-5/ [Último acceso: 27 de agosto de 2015]

[9]Andreas Bærentzen et al. 2006. Single-pass wireframe rendering. In ACM SIGGRAPH 2006 Sketches (SIGGRAPH '06). ACM, New York, NY, USA, , Article 149

[10]António Ramires Fernandes; Bruno Oliveira, OpenGL Insights chapter GPU Tessellation: We Still Have a LOD of Terrain to Cover, CRC Press, 2012, pp. 145–162.

[11]Gavin S P Miller. 1986. The definition and rendering of terrain maps. SIGGRAPH Comput. Graph (Agosto 1986)

[12]Markus Persson, The Word of Notch [en línea] http://notch.tumblr.com/post/3746989361/terrain-generation-part-1 [Último acceso: 30 de agosto de 2015]